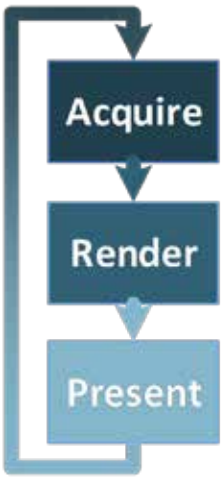
秘密のない API: Vulkan* への実践的なアプローチ - パート 1"

この記事は、インテル® デベロッパー・ゾーンに公開されている「API without Secrets: The Practical Approach to Vulkan* - Part 1」の日本語参考訳です。

サンプル 1: フレームリソース数



単純な課題であり、最も一般的なアプリケーションのレイアウトから説明します。画面に物体をレンダリングする Vulkan* アプリケーションは、次のような構造を持っています。

1. スワップ・チェーン・イメージを取得する。
2. 取得したイメージにレンダリングする。
3. 画面にイメージを出力する。
4. プロセスを繰り返す。

この構造を設定して使用するには、一連のリソースを準備する必要があります。レンダリングしたり、ジョブを実行するには、少なくとも 1 つのコマンドバッファーが必要です。これは、取得したイメージにシーンをレンダリングするために使用します。しかし、取得したイメージがレンダリングに使用できるようになり、プレゼンテーション・エンジンから許可が下りるまで、レンダリング処理を開始することはできません。これは、イメージの取得時に指定するセマフォーやフェンスによって制御されます。Vulkan では、フェンスは CPU (アプリケーション) と GPU (グラフィックス・ハードウェア) の同期に使用され、セマフォーは内部で GPU の同期に使用されます。CPU 側で待機することは推奨しません。通常、レンダリング・パイプラインでストールが発生します。GPU に十分なコマンドを供給しないため、アプリケーションがブロックされるのは望ましくありません。そのため、できるだけセマフォーの使用を推奨します。

これで、アニメーションの単一フレームのレンダリングに必要なリソースの 2 つ (コマンドバッファー、そして GPU がスワップチェーンから取得したイメージを使用できるようになると通知するセマフォー) は揃いました。このコマンドバッファーを記録して送信し、セマフォーを提供します。セマフォーは、カラーデータをイメージに出力できるようになるとレンダリング・パイプラインに通知します。プレゼンテーション処理とレンダリング処

理も同期する必要があるため、まだイメージを画面に出力することはできません。レンダリングが完了するまでイメージを出力できないため、別のセマフォーが必要になります。このセマフォーは、GPU が送信されたコマンドバッファーの処理を完了すると通知を受け取ります。この方法でイメージを表示できるようになったらプレゼンテーション・エンジンに通知します。

上記の 3 つのリソース (コマンドバッファーと 2 つのセマフォー) がすべてではありません。その他のリソースも必要です。通常、アプリケーションのクリティカルな部分 (レンダリング・ループなど) で使用されるリソースは、できるだけ再利用したいと考えます。リソースの作成と破棄にはコストと時間がかかるため、これをフレームごとに行うのは良い選択とは言えません。一連の作成/破棄が問題にならない場合もありますが、コードを複雑にせずに回避できる場合は試してみるべきです。

通常、レンダリング処理でリソースを使用している間は、そのリソースを変更できません。例えば、ハードウェアがコマンドバッファーに記録されているコマンドを読み取っている間は、コマンドバッファーを変更できません。セマフォーを待機していた操作によって通知が解除済みかどうか分からなければ、そのセマフォーを再利用することはできません。リソースは、ハードウェアが確実にもうそれを必要としないことが分かっている場合のみ変更できます。選択したコマンドの処理が完了しているかどうかチェックするには、フェンスを使用する必要があります。

通常、上記のリソースをすべて再利用して、アニメーションのより多くのフレームを準備し、コマンドバッファーを再度記録し、2 つのセマフォーを再利用したいと考えます。リソースを破棄して、フレームごとにそれに対応するものを作成することは望ましくありません。しかし、使用中のリソースは、破棄したいと思っても破棄できません。そのため、フェンスが必要になります。これで、アニメーションの単一フレームのレンダリングに必要なリソースの最小セットが得られます。

1. 少なくとも 1 つの **コマンドバッファー**。このバッファーは、レンダリング・コマンドを記録して、グラフィックス・ハードウェアへ送信するのに必要です。
2. **イメージの準備完了 (イメージ取得済み) セマフォー**。このセマフォーは、プレゼンテーション・エンジンから通知を受け取り、レンダリングとスワップ・チェーン・イメージの取得の同期に使用されます。
3. **レンダリング完了 (出力準備完了) セマフォー**。このセマフォーは、コマンドバッファーの処理が完了すると通知を受け取り、プレゼンテーションとレンダリング処理の同期に使用されます。
4. **フェンス**。フェンスは、フレーム全体のレンダリングの完了を示し、指定されたフレームのリソースが再利用可能になるとアプリケーションに通知します。

ここまでで、単一フレームを準備してレンダリングするには、少なくとも 4 つの Vulkan* リソースが必要なことが分かりました。もちろん、その他のリソースを使用することもできます。(深度テストに使用するため) 深度の添付ファイルとしてイメージが必要になることが良くありますが、以前のイメージが使用されている間に別のイメージを使用したい場合、そのイメージもフレーム・リソース・セットに含めることができます。また、レンダリングに使用するフレームバッファーも追加できます。各フレームは異なるスワップ・チェーン・イメージにレンダリングします (プレゼンテーション・エンジンによってどのイメージが提供されるかは不明です)。そして、そのイメージからフレームバッファーを作成する必要があります。フレームバッファーをフレームリソースに含めることで、コードを簡素化し、保守を容易にできます。

実行する操作の種類に応じて必要なリソースは異なります。上記の 4 つのリソース (2 つのセマフォー、コマンドバッファー、フェンス) は、フレームのレンダリング処理を管理するのに必要な絶対最小要件です。

課題

効率良くフレームを準備して画面に表示するには、フレーム・リソース・セットがいくつ必要でしょうか? 直感的に 1 セットでは不十分であることが分かります。なぜならば、イメージを取得し、コマンドバッファーを記録して送信し、イメージを出力するからです。これらの操作は、セマフォーを使用して内部で同期されます。アニメーションの別のフレームの準備を開始したくても、コマンドバッファーの処理が完了し、フェンスが通知されるまで待機する必要があります。送信されたコマンドの複雑さに応じて、(GPU が処理により多くの時間を必要とするため) 待機時間は長くなります。レンダリングの終了後にのみ、別のフレームのコマンドバッファーの準備を開始することができます。準備中、GPU はアイドル状態となり、新しいコマンドが送信されるまで待機します。これにより、CPU と GPU の両方が効率良く動作せず、多くの時間がアイドルに費やされます。

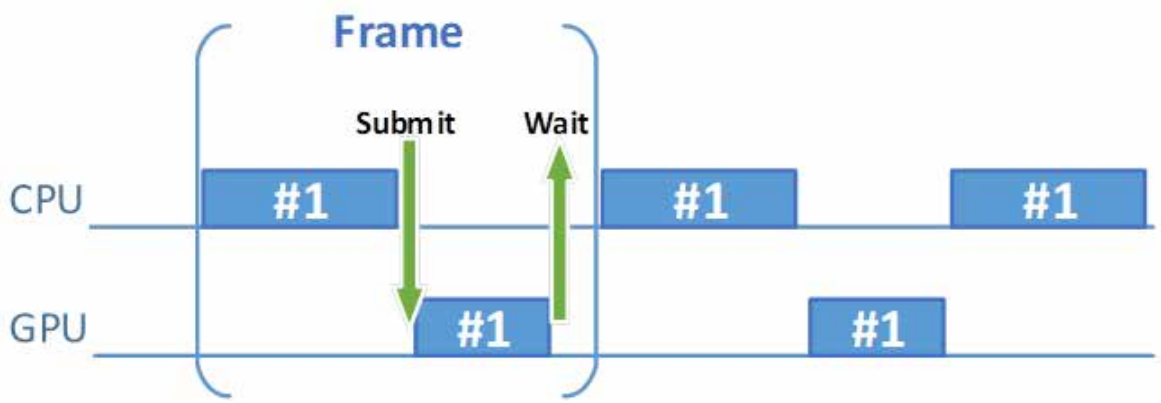


図 1. GPU と CPU のアイドル状態を示すギャップ

時間を無駄にしないようにするためには、アニメーションの別のフレームの準備に使用できる別のリソースセットが必要です。

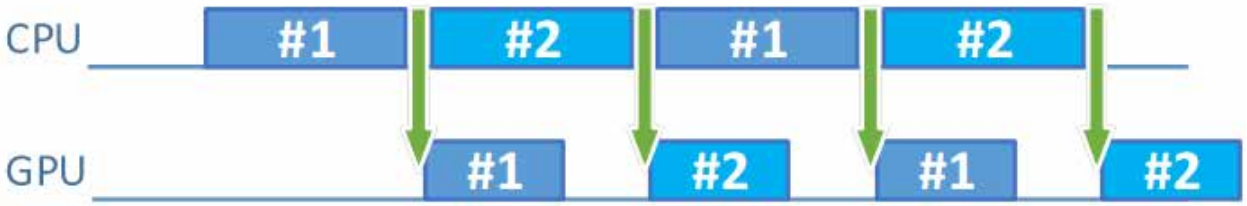


図 2. アイドル状態を軽減して効率を向上

まず、最初のフレームを準備して送信します。そして、別のフレーム・リソース・セットを使用して、その直後に別のフレームの準備を開始します。そのため、最初のフレームのコマンドの処理が完了するまで待機する必要がありません。おそらく、2 つ目のフレームの準備が終わるころには、最初のフレームは (GPU 依存でない限り) レンダリングが完了しているでしょう。

次の 3 つ目のフレームはどうすべきでしょうか? 最初のフレームが終了するまで待機して、そのリソースを再利用すべきでしょうか? それとも、3 つ目のフレーム・リソース・セットを用意すべきでしょうか?フレームリソースはいくつ必要でしょうか?答えは、サンプルプログラムが与えてくれるでしょう。

サンプルプログラム

この記事のサンプルプログラムは、複数のテクスチャー付きクワッドを含むシンプルなシーンを表示します。各クワッドは 3,200 個のトライアングル (コードで簡単に調整可能) で構成されているため、単純に見えますが、シーンには多数のバーテックスがあります。これは意図的なもので、コマンドバッファーの生成時間に影響することなく、シーンの複雑さを簡単に調整できるようにしています。初期設定では、100 クワッド (320,000 個のトライアングル) に設定されていますが、クワッドの数は変更できます。このサンプルプログラムを使って、フレーム・リソース・セットの数がレンダリング・パフォーマンスに与える影響をテストできます。追加の計算に費やされる CPU 時間も特定できます。

図 3. 複数のテクスチャー付きクワッドを含むサンプルプログラム
(各クワッドには 3,200 のトライアングルがある)

以下は、サンプルプログラムのレンダリング・ループの最も一般的な構造です。

1. フェンスで次の使用済みフレーム・リソース・セットを待機する。
2. スワップ・チェーン・イメージを取得する。
3. 送信前の計算を実行する (現在のフレームのコマンドバッファーの記録に影響するワークのシミュレーション)。
4. コマンドバッファーを記録して送信する。
5. その他の計算を実行する (送信後に実行されるワークのシミュレーション)。
6. GUI を描画する。(指定されたフレーム・リソース・セットからフェンスに通知を送信します。)
7. スワップ・チェーン・イメージを出力する。

パラメーター

以下は、サンプルプログラムで使用されるパラメーターです。

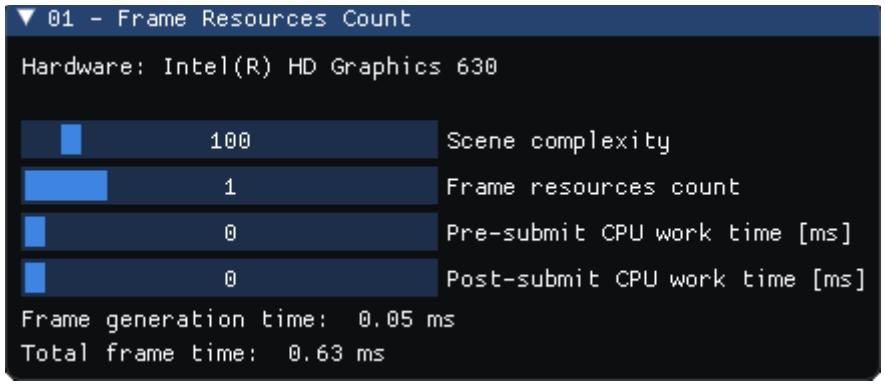


図 4. サンプルプログラムのパラメーター

- **Hardware (ハードウェア)**: アプリケーションを実行するグラフィックス・ハードウェアの名前。
- **Scene complexity (シーンの複雑さ)**: シーンに表示されるクワッドの数を調整します。このパラメーターを使用して GPU が各フレームで処理する頂点の数を変更し、アプリケーション全体のパフォーマンスを向上または低下させます。
- **Frame resources count (フレームリソース数)**: レンダリングに使用されるフレーム・リソース・セットの数を指定します。
- **Pre-submit CPU work time (送信前の CPU 作業時間)**: 現在記録されているコマンドバッファーに影響する計算 (シーンに関連する可視性のカリングなど) に費やされた CPU 時間 (ミリ秒) をシミュレーションします。これらの計算は、コマンドバッファーの送信*前*に実行されます。
- **Post-submit CPU work time (送信後の CPU 作業時間)**: アニメーションの現在のフレームに直接関係のない計算 (人工知能 (AI) の計算、ネットワーク関連の処理、アニメーションの次のフレームに影響する計算など) に費やされた CPU 時間 (ミリ秒) をシミュレーションします。これらの計算は、コマンドバッファーの送信*後*、出力の前に実行されます。
- **Frame generation time (フレーム生成時間)**: フレームのデータ生成にかかった時間。コマンドバッファーの記録と送信にかかった時間、およびシーンと AI 計算の両方が含まれます。
- **Total frame time (合計フレーム時間)**: 終了フェンスを待機する直前からスワップ・チェーン・イメージの出力後まで、アニメーションの単一フレームの準備に必要な合計時間。

実証

ここでは、サンプルプログラムで検証できることを説明し、出力された値を解釈する方法を示します。検証を容易にするため、最初に **Scene complexity** パラメーターを変更して、アプリケーションのパフォーマンスを調整します。60fps でゲームのようなパフォーマンスが得られるようにします。**Total frame time** 値が示すように、フレームの生成 (CPU) とレンダリング (GPU) には、合わせて約 16 ミリ秒 (ms) かかります。しかし、**Frame generation time** 値が示すように、実際のフレーム出力にはわずかな時間しかかかりません。合計時間のほとんどはフェンスの待機に費やされています。**Frame resources count** を増やしたらどうなるでしょうか? (低速な CPU のコンピューターでアプリケーションを実行している場合を除いて) あまり変わりありません。パフォーマンスがわずかに向上するかもしれませんが、大きな変化は見られません。なぜでしょうか? 最初のコマンドバッファーの生成時間がごくわずかだからです。これは、アプリケーションが GPU 依存であることを示しています。

実際のアプリケーションでは、コマンドバッファーの生成にかなりの時間がかかります。表示されるオブジェクトを確認する必要があったり、物理計算を実行したり、バックグラウンドでデータ・ストリーミングを行っている可能性があります。このような場合、**Pre-submit CPU work time** パラメーターが役立ちます。このパラメーター値を増やすことで、より大きな CPU ワークロードのシミュレーションが可能です。サンプルプログラムのフレーム生成 (レンダリング) 時間は約 16ms であるため、**Scene calculations time** パラメーターを 14 - 15ms に増やします (GPU がシーン全体をレンダリングするのにかかる時間よりも長くならないようにします)。1 つのフレーム・リソース・セットのみを使用するとどうなりますか? アプリケーションのパフォーマンスが大幅に低下します。これは、連続するコマンドバッファーの送信間隔が長くなったことで、GPU がコマンドを待機する時間が増えたためです。最初のフレーム生成 (コマンドバッファーの記録) 時間は、次に示すように非常に短いものでした。

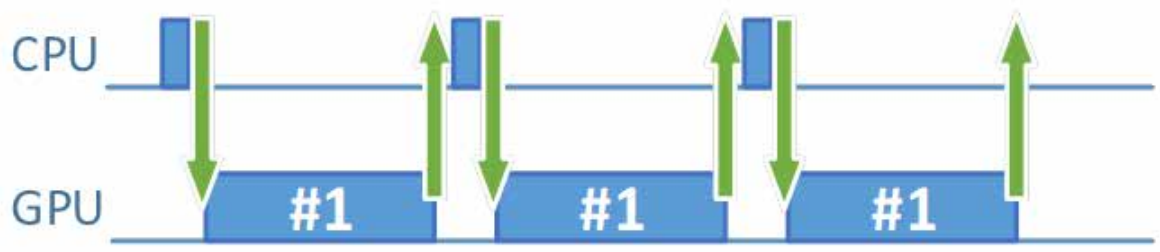


図 5. コマンドバッファーの生成時間が短い場合

Pre-submit CPU work time パラメーターの値を増やしたところ、アイドル状態で待機する時間が増えました (GPU 処理タイムライン上のギャップが大きくなったことが分かります)。

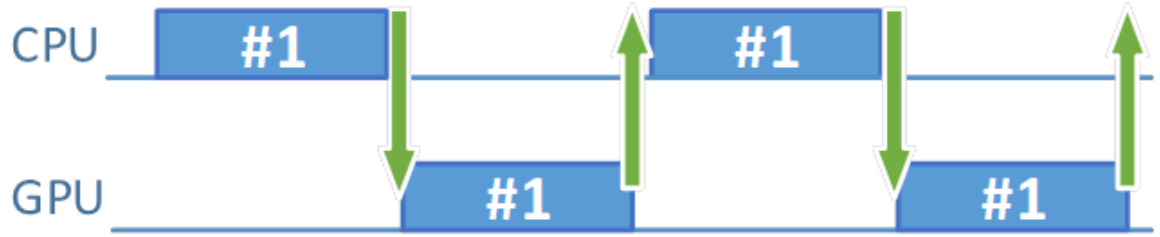


図 6. コマンドバッファーの生成時間を増やした場合 (より大きな CPU ワークロードのシミュレーション)

次に、**Frame resources count** パラメーターの値を 1 から 2 に変更してみます。どうなりますか? アプリケーションのパフォーマンスが最初の 60 fps に戻ります。CPU でより多くの処理を実行して (処理にはまだ約 14 - 15ms かかります)、パフォーマンスに影響を与えることなく、アプリケーションがより多くのことを行えるようにします。調整により、実行時間全体をより効率的に使用できるようになります。

図 7. フレームリソース数が増えたことでアイドル時間が大幅に減少

Pre-submit CPU work time パラメーターを 16ms よりも大きくしたらどうなるでしょうか? この時点でアプリケーションは CPU 依存であるため、パフォーマンスが低下します。

図 8. Pre-submit CPU work time が増えると CPU がボトルネックになる

フレーム・リソース・セットの数を増やしても効果はありません。GPU にコマンドを供給できなかったり、イメージを素早く取得し出力できなければ、CPU がボトルネックになります。上記のイメージでは、フェンスを待機する必要はありません (ただし、形式上フェンスの状態をチェックする必要があります)。これは、2 つ目のフレーム・リソース・セットのコマンドバッファーの記録が終了する前に、GPU は 1 つ目のフレーム・リソース・セットのコマンドバッファーの処理を完了しているためです。このような場合、CPU で実行される計算のみ簡素化できます。CPU 側の計算を簡素化しない場合は、より高速な CPU が必要です。

フレーム・リソース・セットの数を増やすと、CPU と GPU の待機時間が最小になります。その時間を最適に使用できなければ、アプリケーション全体のパフォーマンスは向上しません。

調整可能な別のパラメーターに **Post-submit CPU work time** があります。このパラメーターを調査するには、すべてのパラメーターを初期値にリセットします。**Frame resources count** を 1 にし、2 つの CPU work time を 0 に設定します。**Scene complexity** パラメーターは、シーンが 60fps でレンダリングされるように設定します。その後、**Post-submit CPU work time** パラメーターの値を 14 - 15ms に増やします。どうなりますか? 何も起こりません。レンダリング・パフォーマンスに変化はありません。なぜでしょうか? 以前の実験では、**Pre-submit CPU work time** パラメーターの値を増やしたところ、パフォーマンスは低下しました。今回はどうして低下しなかったのでしょうか? **Pre-submit CPU work time** は、送信前に実行される計算をシミュレーションします。**Post-submit CPU work time** は、送信後に実行される計算をシミュレーションします。以前は、送信後の時間のほとんどはフェンスの待機に費やされており、無駄な時間でした。今回は、より建設的なものに時間が費やされています。

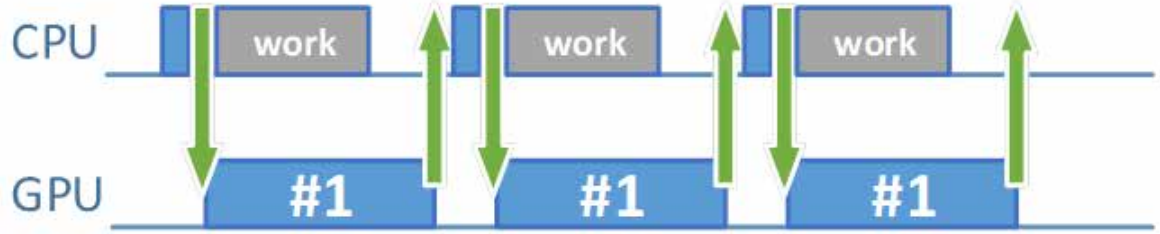


図 9. Post-submit CPU work time パラメーターを増やしてもパフォーマンスへの影響はわずか

もちろん、**Post-submit CPU work time** を大幅に増やせば、再び CPU 依存となり、パフォーマンスが低下します。

まとめ

結局のところ、どうすべきでしょうか? フレーム・リソース・セットの数を増やす必要はあるのでしょうか? あるいは、適切な時間で計算を実行できるようにアプリケーションを設計するべきでしょうか?

少なくとも 2 つのフレーム・リソース・セットが必要です。実験では、フレーム・リソース・セットを 3 つに増やしてもパフォーマンスは (全く) 向上しませんでした。

CPU ワークロードがすべてのフレームに均等に分散されていない状況は珍しくありません。一部のフレームは短時間で生成することができる一方、別のフレームではデータの準備により時間がかかる可能性があります。これは、CPU と GPU ワークのバランスをとることとほぼ同じです。そのような状況では、3 つのフレーム・リソース・セットを使用することで、3 つ目のフレーム・リソース・セットがフレーム生成時間の違いを補い、フレームレートが安定するでしょう (これを検証するには別のサンプルプログラムが必要になるでしょう)。ここで行った実験では、フレーム生成とレンダリング時間が安定している場合は、2 つのフレーム・リソース・セットで十分であることを示しています。

CPU と GPU 処理時間のバランスをとることは難しいため、3 つのフレーム・リソース・セットを使用することを推奨します。

フレームリソースの数とスワップ・チェーン・イメージの数を混同しないでください。この 2 つは関連している必要はありません。設計上、この 2 つを同じ値にすることはできますが、一般にそうする必要はありません。同じ生成パラメーターを使用しても、あるスワップチェーンに対して作成されるイメージ数は、ドライバーによって異なることがあります。スワップチェーンの作成中、必要なイメージの最小数を指定しますが、実装 (ドライバー) はそれよりも多くのイメージを作成することがあります。そのため、フレームリソースの数とスワップ・チェーン・イメージの数を関連付けると、さまざまなプラットフォームでアプリケーションの動作が異なる可能性があり、特にメモリー使用量が多い場合それが顕著になります。

CPU がコマンドバッファーの記録とその他の計算に費やす時間を考慮します。一般に、CPU がフレームデータの準備に費やす時間と GPU がデータの処理に費やす時間のバランスをとるべきです。GPU が高速で CPU がデータを生成するよりもはるかに速くシーンをレンダリングできる場合、CPU 依存となり、グラフィックス・ハードウェアの性能を最大限に活用できません。一方で、CPU がレンダリング用のデータを簡単に作成できても GPU が常にビジー状態の場合、GPU 依存となります。この場合、GPU からさらなるパフォーマンスを引き出すことは困難 (不可能でなければ) ですが、より高精度の物理計算や AI 計算に追加の時間を費やすことができます。また、モバイルデバイスをターゲットにする場合、CPU ワークロードを小さくすることで、電力消費を軽減できます。

CPU 側で計算が実行される場合はどうでしょうか? サンプルプログラムは、これが非常に重要であることを示しています。効率良く管理された CPU ワークロードを使用することで、CPU で十分な計算を行いつつ、GPU の潜在能力を活用することができます。しかし実際には、アプリケーションをそのように設計するのは難しいかもしれません。特に、ネットワーク、サウンド管理、ストリーミング・データ、バックグラウンドで処理を実行する複数のスレッドの同期などのタスクを含むゲームを作成する場合、これらの処理がいつ発生するのかを正確に計画するのは難しいことがあります。単一のフレーム・リソース・セットでは、フレーム生成時間のわずかな変化が顕著になり、フレームレートが滑らかでなくなります。さらに、唯一のコマンドバッファーの記録には、おそらくサンプルプログラムよりもはるかに長い時間がかかるでしょう。少なくとも 2 つのフレーム・リソース・セットを用意することで、これらの問題を回避し、フレームレートを滑らかにすることができます。

これは、デバッグツールとしても興味深いでしょう。フレーム・リソース・セットの数を減らした場合、アプリケーション・パフォーマンスへの影響がどのように変化するのか確認できます。大きな変化が見られない場合は、CPU の処理能力を最大限に利用していないことを示しており、パフォーマンス向上の可能性があります。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。